# ECE 438: Computer Design
# Electrical and Computer Engineering
# University of New Mexico

Instructor: Andrew Targhetta

# Lab 2

# The Instruction and ALU Decoders

# Due: March 12th, 2020

## Introduction

The objective of the second laboratory exercise is to describe the decode logic using an HDL. As with the previous lab, this lab is VHDL centric, but you are welcome to use Verilog.

## Background

The decode logic within our processor takes as input the *Opcode* and *Function* fields from the MIPS instruction and outputs a number of control signals that orchestrate the flow of data through the datapath. To simplify the decode logic and remain consistent with lecture material, we will create two components for decoding, namely the **Instruction Decode Unit** and the **ALU Decode Unit**. Figure 1 depicts the single cycle microarchitecture with both decode logic components and their associated output control signals. For now, we will support the following instructions:

- Arithemtic — ADDU, ADDIU, SUBU
- Logical — AND, ANDI, NOR, OR, ORI, XOR, XORI
- Shift — SLL, SRL, SRA

- Comparison — SLTU, SLTIU
- Loads and stores — LW and SW
- Branch Equal and Jump — BEQ and J
- Load Upper Immediate — LUI

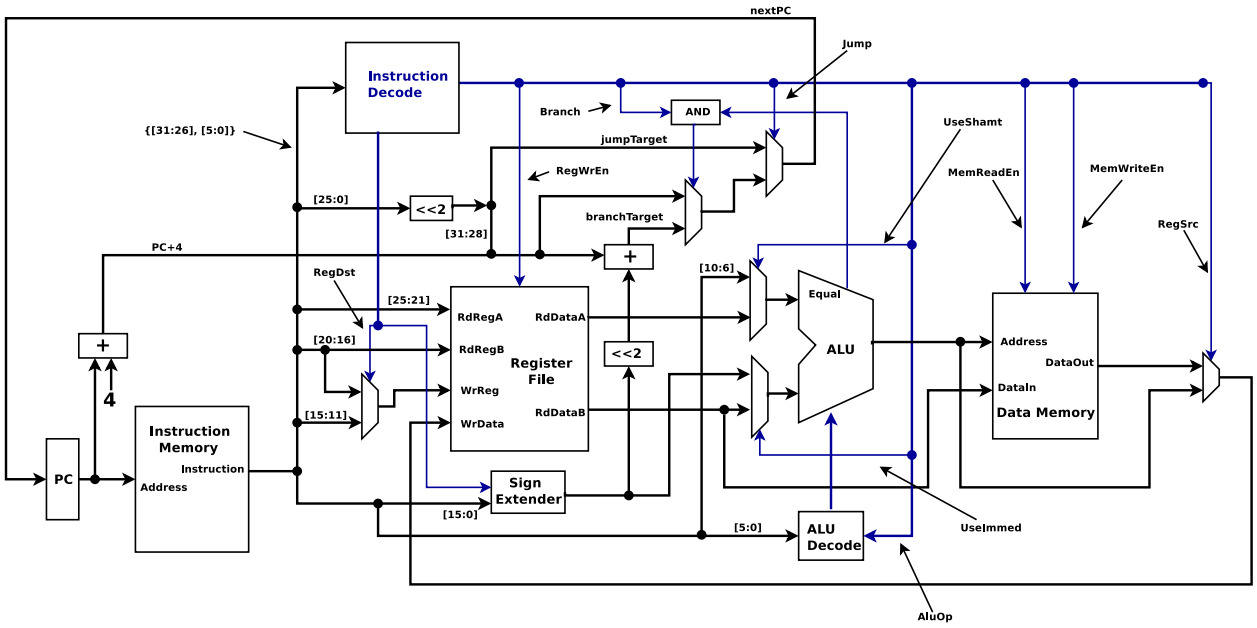


Figure 1: Single-cycle MIPS Block Diagram

## Procedure

1. Describe the ALU decode unit in VHDL. The **ALU Decode Unit** allows further decoding of an R-type instruction with the *Function* field from the MIPS instruction. The process of decoding an instruction into a control code that the ALU can understand (Table 1) is split up into two blocks. The first block is contained within the instruction decode unit and will be addressed shortly. The second block is the ALU decode unit shown in Figure 1, which takes in an ALU op code from the main decode unit and the function field from the instruction. The ALU control logic should asynchronously function as follows:

| Operation | ALU Control Line |
|---|---|
| AND | 0000 |
| OR | 0001 |
| SLL | 0011 |
| SRL | 0100 |
| ADDU | 1000 |
| SUBU | 1001 |
| XOR | 1010 |
| SLTU | 1011 |
| NOR | 1100 |
| SRA | 1101 |
| LUI | 1110 |

Table 1: Arithmetic Logic functions and associated control codes

- When *AluOp* is not equal to "1111," *AluCtrl* should be set to *AluOp*.
- When *AluOp* is equal to "1111," the function field should be used to determine *AluCtrl*.

For R-type instructions, the instruction decode unit simply sets the *AluOp* to "1111" and the ALU decode unit will interpret the arithmetic or logic function needed to carry out the instruction and pass this information on to the ALU. However, when an instruction other than an R-type instruction is decoded, the instruction decode unit must specify the correct ALU control code on *AluOp*.

**Helpful Hints:**

- Describe the ALU decode logic in VHDL using the following entity:

```
entity AluDecode is
port( AluOp: in std_logic_vector(3 downto 0);
      Funct: in std_logic_vector(5 downto 0);
      AluCtrl: out std_logic_vector(3 downto 0)
);
end AluDecode;
```

- CASE statements can be nested within IF ELSE statements within PROCESS blocks.

2. Describe the Instruction Decode Unit in VHDL. The **Instruction Decode Unit** essentially takes as input the *Opcode* field from the MIPS instruction and decodes it into various control signals that will orchestrate the flow of data. Use the skeleton code below to guide you through the process:

```
----------------------------------------------------
-- This component describes the instruction decode unit
-- in our MIPS processor.
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

----------------------------------------------------

entity InstrDecode is

port( --6-bit op field of instruction
      Opcode: in std_logic_vector(5 downto 0);
      --6-bit funct field of instruction
      Funct: in std_logic_vector(5 downto 0);
      -- asserted when shamt field is input to ALU
      UseShamt: out std_logic;
      -- asserted when the immediate field is input to ALU
      UseImmed: out std_logic;
      -- asserted when immed needs to be sign extended
      SignExtend: out std_logic;
      -- asserted when instruction is a jump
      Jump: out std_logic;
      -- asserted when instruction is a branch
      Branch: out std_logic;
      -- Determines ALU operation, see ALU decode unit
      AluOp: out std_logic_vector(3 downto 0);
      -- selects between rt and rd for the register destination
      RegDst: out std_logic;
      -- asserted when reading from memory (loads!)
      MemRdEn: out std_logic;
      -- asserted when writing to memory (stores!)
      MemWrEn: out std_logic;
      -- selects the source for writing into register file
      RegSrc: out std_logic;
      -- asserted when writing into register file
      RegWrEn: out std_logic
      -- that's all folks!
);
end InstrDecode;

----------------------------------------------------

architecture InstrDecode of InstrDecode is

begin
    --One big ol' case statement makes this description easy!
    process(Opcode, Funct) -- output depends on op field and funct
```

```
50      begin
            case Opcode is
52              -- R-type instruction
                -- For our small instruction set, the R-type instruction
54              -- is the only instruction where we have to look at the
                -- function field...
56              when b"000000" =>
                    -- SLL, SRL, and SRA use the shamt field
58                  if(Funct=b"000000" or Funct=b"000010" or Funct=b"000011") then
                        UseShamt <= '1';-- using shamt
60                  else
                        UseShamt <= '0'; --not using shamt
62                  end if;
                    UseImmed <= '0';-- nor immediate
64                  SignExtend <= '-'; -- doesn't matter
                    Jump <= '0'; --don't jump!
66                  Branch <= '0'; -- don't branch either
                    AluOp <= b"1111"; --use function field
68                  RegDst <= '1'; --rd specifies destination
                    MemRdEn <= '0'; --leave memory alone
70                  MemWrEn <= '0';
                    RegSrc <= '1'; --use ALU ouput
72                  RegWrEn <= '1'; --writing into register file
                -- Fill in the rest of the supported instructions
74              -- to avoid describing latches
                when others =>
76                  -- don't care because invalid instruction!
                    UseShamt <= '-';
78                  UseImmed <= '-';
                    SignExtend <= '-';
80                  Jump <= '0'; --don't jump!
                    Branch <= '0'; -- don't branch either
82                  AluOp <= (others => '-');
                    -- fill in the rest here!
84                  MemRdEn <= '0'; --leave memory alone
                    MemWrEn <= '0';
86                  -- finish this, but don't allow state to change
                    -- when an invalid instruction is encountered!
88                  -- I'm going to test for this, so get it right...
            end case;
90      end process;
        -- was that so hard?!?
92  end InstrDecode;
```

# 1 Deliverables

1. Email all HDL source files with comments to `adtargh@unm.edu` .

   *Note: Code submitted without adequate comments will not be graded!*

2. Answer the following questions:

   (a) Is the instruction decode unit combinational logic or sequential logic? Describe your reasoning.

   (b) Is the ALU decode unit combinational logic or sequential logic? Describe your reasoning.

   (c) How might you modify the instruction decode HDL to incorporate the functionality of the ALU decode unit?